assignment1

February 15, 2020

1 [COM4513-6513] Assignment 1: Text Classification with Logistic Regression

1.0.1 Instructor: Nikos Aletras

The goal of this assignment is to develop and test two text classification systems:

- **Task 1:** sentiment analysis, in particular to predict the sentiment of movie review, i.e. positive or negative (binary classification).
- **Task 2:** topic classification, to predict whether a news article is about International issues, Sports or Business (multiclass classification).

For that purpose, you will implement:

- Text processing methods for extracting Bag-Of-Word features, using (1) unigrams, bigrams and trigrams to obtain vector representations of documents. Two vector weighting schemes should be tested: (1) raw frequencies (**3 marks; 1 for each ngram type**); (2) tf.idf (**1 marks**).
- Binary Logistic Regression classifiers that will be able to accurately classify movie reviews trained with (1) BOW-count (raw frequencies); and (2) BOW-tfidf (tf.idf weighted) for Task 1.
- Multiclass Logistic Regression classifiers that will be able to accurately classify news articles trained with (1) BOW-count (raw frequencies); and (2) BOW-tfidf (tf.idf weighted) for Task 2.
- The Stochastic Gradient Descent (SGD) algorithm to estimate the parameters of your Logistic Regression models. Your SGD algorithm should:
  - Minimise the Binary Cross-entropy loss function for Task 1 (**3 marks**)
  - Minimise the Categorical Cross-entropy loss function for Task 2 (**3 marks**)
  - Use L2 regularisation (both tasks) (**1 mark**)
  - Perform multiple passes (epochs) over the training data (**1 mark**)
  - Randomise the order of training data after each pass (**1 mark**)
  - Stop training if the difference between the current and previous validation loss is smaller than a threshold (**1 mark**)
  - After each epoch print the training and development loss (**1 mark**)
- Discuss how did you choose hyperparameters (e.g. learning rate and regularisation strength)? (**2 marks; 0.5 for each model in each task**).

- After training the LR models, plot the learning process (i.e. training and validation loss in each epoch) using a line plot (**1 mark; 0.5 for both BOW-count and BOW-tfidf LR models in each task**) and discuss if your model overfits/underfits/is about right.
- Model interpretability by showing the most important features for each class (i.e. most positive/negative weights). Give the top 10 for each class and comment on whether they make sense (if they don't you might have a bug!). If we were to apply the classifier we've learned into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain? (**2 marks; 0.5 for BOW-count and BOW-tfidf LR models respectively in each task**)

1.0.2 Data - Task 1

The data you will use for Task 1 are taken from here: `http://www.cs.cornell.edu/people/pabo/movie-review-data/` and you can find it in the `./data_sentiment` folder in CSV format:

- `data_sentiment/train.csv`: contains 1,400 reviews, 700 positive (label: 1) and 700 negative (label: 0) to be used for training.
- `data_sentiment/dev.csv`: contains 200 reviews, 100 positive and 100 negative to be used for hyperparameter selection and monitoring the training process.
- `data_sentiment/test.csv`: contains 400 reviews, 200 positive and 200 negative to be used for testing.

1.0.3 Data - Task 2

The data you will use for Task 2 is a subset of the AG News Corpus and you can find it in the `./data_topic` folder in CSV format:

- `data_topic/train.csv`: contains 2,400 news articles, 800 for each class to be used for training.
- `data_topic/dev.csv`: contains 150 news articles, 50 for each class to be used for hyperparameter selection and monitoring the training process.
- `data_topic/test.csv`: contains 900 news articles, 300 for each class to be used for testing.

1.0.4 Submission Instructions

You should submit a Jupyter Notebook file (assignment1.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex`).

You are advised to follow the code structure given in this notebook by completing all given funtions. You can also write any auxilliary/helper functions (and arguments for the functions) that you might need but note that you can provide a full solution without any such functions. Similarly, you can just use only the packages imported below but you are free to use any functionality from the Python Standard Library, NumPy, SciPy and Pandas. You are not allowed to use any third-party library such as Scikit-learn (apart from metric functions already provided), NLTK, Spacy, Keras etc..

Please make sure to comment your code. You should also mention if you've used Windows (not recommended) to write and test your code. There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1-scores around 80% or higher. The quality of the analysis of the results is as important as the accuracy itself.

This assignment will be marked out of 20. It is worth 20% of your final grade in the module.

The deadline for this assignment is **23:59 on Fri, 20 Mar 2020** and it needs to be submitted via MOLE. Standard departmental penalties for lateness will be applied. We use a range of strategies to detect unfair means, including Turnitin which helps detect plagiarism, so make sure you do not plagiarise.

```
In [4]: import pandas as pd
        import numpy as np
        from collections import Counter
        import re
        import matplotlib.pyplot as plt
        from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
        import random

        # fixing random seed for reproducibility
        random.seed(123)
        np.random.seed(123)
```

1.1 Load Raw texts and labels into arrays

First, you need to load the training, development and test sets from their corresponding CSV files (tip: you can use Pandas dataframes).

```
In [2]: # fill in your code...
```

If you use Pandas you can see a sample of the data.

```
In [3]: data_tr.head()
```

```
Out[3]:                                                text  label
        0  note : some may consider portions of the follo...      1
        1  note : some may consider portions of the follo...      1
        2  every once in a while you see a film that is s...      1
        3  when i was growing up in 1970s , boys in my sc...      1
        4  the muppet movie is the first , and the best m...      1
```

The next step is to put the raw texts into Python lists and their corresponding labels into NumPy arrays:

```
In [4]: # fill in your code...
```

2 Bag-of-Words Representation

To train and test Logisitc Regression models, you first need to obtain vector representations for all documents given a vocabulary of features (unigrams, bigrams, trigrams).

2.1 Text Pre-Processing Pipeline

To obtain a vocabulary of features, you should: - tokenise all texts into a list of unigrams (tip: using a regular expression) - remove stop words (using the one provided or one of your preference) - compute bigrams, trigrams given the remaining unigrams - remove ngrams appearing in less than K documents - use the remaining to create a vocabulary of unigrams, bigrams and trigrams (you can keep top N if you encounter memory issues).

```
In [5]: stop_words = ['a','in','on','at','and','or',
                      'to', 'the', 'of', 'an', 'by',
                      'as', 'is', 'was', 'were', 'been', 'be',
                      'are','for', 'this', 'that', 'these', 'those', 'you', 'i',
                     'it', 'he', 'she', 'we', 'they' 'will', 'have', 'has',
                      'do', 'did', 'can', 'could', 'who', 'which', 'what',
                     'his', 'her', 'they', 'them', 'from', 'with', 'its']
```

2.1.1 N-gram extraction from a document

You first need to implement the `extract_ngrams` function. It takes as input: - `x_raw`: a string corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `vocab`: a given vocabulary. It should be used to extract specific features.

and returns:

- a list of all extracted features.

See the examples below to see how this function should work.

```
In [6]: def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b', s

            # fill in your code...

            return x
```

```
In [7]: extract_ngrams("this is a great movie to watch",
                       ngram_range=(1,3),
                       stop_words=stop_words)
```

```
Out[7]: ['great',
         'movie',
         'watch',
         ('great', 'movie'),
         ('movie', 'watch'),
         ('great', 'movie', 'watch')]
```

```
In [8]: extract_ngrams("this is a great movie to watch",
                       ngram_range=(1,2),
                       stop_words=stop_words,
                       vocab=set(['great',  ('great','movie')]))
```

```
Out[8]: ['great', ('great', 'movie')]
```

Note that it is OK to represent n-grams using lists instead of tuples: e.g. `['great', ['great', 'movie']]`

2.1.2 Create a vocabulary of n-grams

Then the `get_vocab` function will be used to (1) create a vocabulary of ngrams; (2) count the document frequencies of ngrams; (3) their raw frequency. It takes as input: - `X_raw`: a list of strings each corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `vocab`: a given vocabulary. It should be used to extract specific features. - `min_df`: keep ngrams with a minimum document frequency. - `keep_topN`: keep top-N more frequent ngrams.

and returns:

- `vocab`: a set of the n-grams that will be used as features.
- `df`: a Counter (or dict) that contains ngrams as keys and their corresponding document frequency as values.
- `ngram_counts`: counts of each ngram in vocab

Hint: it should make use of the `extract_ngrams` function.

```
In [9]: def get_vocab(X_raw, ngram_range=(1,3), token_pattern=r'\b[A-Za-z][A-Za-z]+\b', min_df=

            # fill in your code...

            return vocab, df, ngram_counts
```

Now you should use `get_vocab` to create your vocabulary and get document and raw frequencies of n-grams:

```
In [10]: vocab, df, ngram_counts = get_vocab(X_tr_raw, ngram_range=(1,3), keep_topN=5000, stop_
         print(len(vocab))
         print()
         print(list(vocab)[:100])
         print()
         print(df.most_common()[:10])
```

```
5000

['manages', 'questions', 'covered', 'body', 'ron', 'flair', 'drunken', 'approach', 'etc', 'allo

[('but', 1334), ('one', 1247), ('film', 1231), ('not', 1170), ('all', 1117), ('movie', 1095),
```

Then, you need to create vocabulary id -> word and id -> word dictionaries for reference:

```
In [11]: # fill in your code...
```

Now you should be able to extract n-grams for each text in the training, development and test sets:

```
In [12]: # fill in your code...
```

2.2 Vectorise documents

Next, write a function `vectoriser` to obtain Bag-of-ngram representations for a list of documents. The function should take as input: - `X_ngram`: a list of texts (documents), where each text is represented as list of n-grams in the `vocab` - `vocab`: a set of n-grams to be used for representing the documents

and return: - `X_vec`: an array with dimensionality Nx|vocab| where N is the number of documents and |vocab| is the size of the vocabulary. Each element of the array should represent the frequency of a given n-gram in a document.

```
In [13]: def vectorise(X_ngram, vocab):

             # fill in your code...

             return X_vec
```

Finally, use `vectorise` to obtain document vectors for each document in the train, development and test set. You should extract both count and tf.idf vectors respectively:

Count vectors

```
In [14]: # fill in your code...

In [15]: X_tr_count.shape

Out[15]: (1400, 5000)

In [16]: X_tr_count[:2,:50]

Out[16]: array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0.,
                 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0.,
                 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                 0., 0.],
                [0., 0., 0., 1., 0., 0., 0., 2., 0., 0., 0., 0., 0., 0., 0., 1.,
                 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
                 0., 0.]])
```

TF.IDF vectors First compute `idfs` an array containing inverted document frequencies (Note: its elements should correspond to your `vocab`)

```
In [17]: # fill in your code...
```

Then transform your count vectors to tf.idf vectors:

```
In [18]: # fill in your code...

In [19]: X_tr_tfidf[1,:50]

Out[19]: array([0.        , 0.        , 0.        , 2.24028121, 0.        ,
                0.        , 0.        , 5.67501654, 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                2.47354289, 0.        , 0.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 0.        ,
                0.        , 0.        , 0.        , 0.        , 2.56209629,
                0.        , 0.        , 0.        , 0.        , 0.        ])

In [ ]:
```

3 Binary Logistic Regression

After obtaining vector representations of the data, now you are ready to implement Binary Logistic Regression for classifying sentiment.

First, you need to implement the `sigmoid` function. It takes as input:

- `z`: a real number or an array of real numbers

and returns:

- `sig`: the sigmoid of `z`

```
In [20]: def sigmoid(z):

             # fill in your code...

             return z

In [21]: print(sigmoid(0))
         print(sigmoid(np.array([-5., 1.2])))
```

```
0.5
[0.00669285 0.76852478]
```

Then, implement the `predict_proba` function to obtain prediction probabilities. It takes as input:

- `X`: an array of inputs, i.e. documents represented by bag-of-ngram vectors ($N \times |vocab|$)
- `weights`: a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_proba`: the prediction probabilities of X given the weights

```
In [22]: def predict_proba(X, weights):

             # fill in your code...

             return preds_proba
```

Then, implement the `predict_class` function to obtain the most probable class for each vector in an array of input vectors. It takes as input:

- `X`: an array of documents represented by bag-of-ngram vectors ($N \times |vocab|$)
- `weights`: a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_class`: the predicted class for each x in X given the weights

```
In [23]: def predict_class(X, weights):

             # fill in your code...

             return preds_class
```

To learn the weights from data, we need to minimise the binary cross-entropy loss. Implement `binary_loss` that takes as input:

- `X`: input vectors
- `Y`: labels
- `weights`: model weights
- `alpha`: regularisation strength

and return:

- `l`: the loss score

```
In [24]: def binary_loss(X, Y, weights, alpha=0.00001):

             # fill in your code...

             return l
```

Now, you can implement Stochastic Gradient Descent to learn the weights of your sentiment classifier. The `SGD` function takes as input:

- `X_tr`: array of training data (vectors)
- `Y_tr`: labels of `X_tr`
- `X_dev`: array of development (i.e. validation) data (vectors)
- `Y_dev`: labels of `X_dev`
- `lr`: learning rate
- `alpha`: regularisation strength
- `epochs`: number of full passes over the training data
- `tolerance`: stop training if the difference between the current and previous validation loss is smaller than a threshold
- `print_progress`: flag for printing the training progress (train/validation loss)

and returns:

- `weights`: the weights learned
- `training_loss_history`: an array with the average losses of the whole training set after each epoch
- `validation_loss_history`: an array with the average losses of the whole development set after each epoch

```
In [25]: def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], loss="binary", lr=0.1, alpha=0.00001, epochs=5

             cur_loss_tr = 1.
             cur_loss_dev = 1.
             training_loss_history = []
             validation_loss_history = []

             # fill in your code...


             return weights, training_loss_history, validation_loss_history
```

3.1 Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

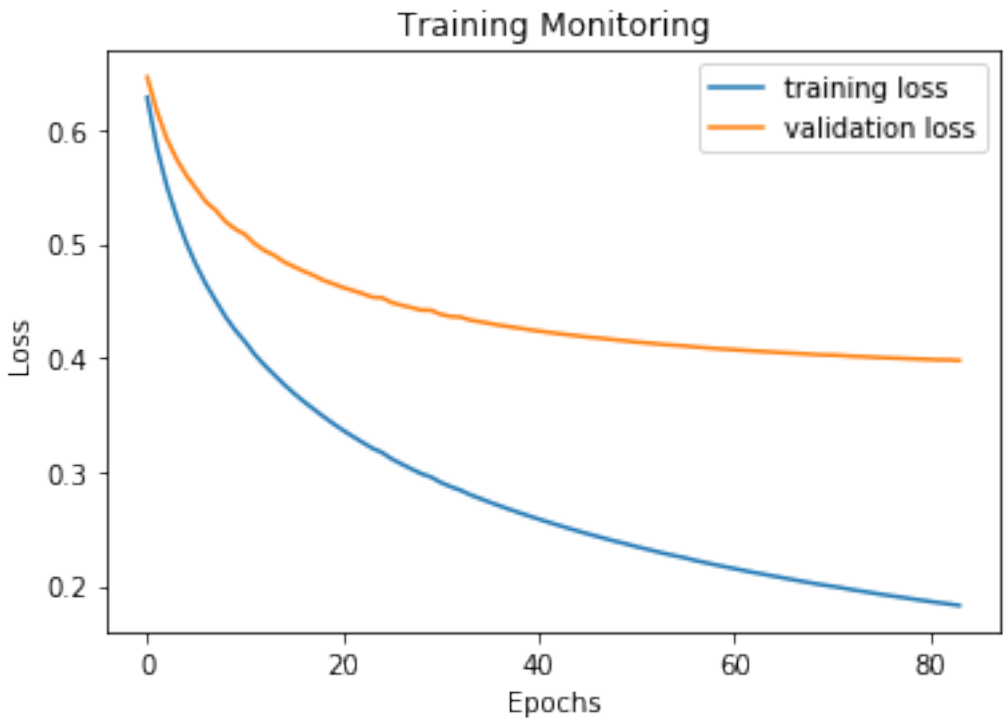
```
In [26]: w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                                     X_dev=X_dev_count,
                                                     Y_dev=Y_dev,
                                                     lr=0.0001,
                                                     alpha=0.001,
                                                     epochs=100)

/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:50: DeprecationWarning: elementwis
```

```
Epoch: 0| Training loss: 0.6281849626714022| Validation loss: 0.6459904270912492
Epoch: 1| Training loss: 0.58394973349301| Validation loss: 0.615917609244642
Epoch: 2| Training loss: 0.5505425987370951| Validation loss: 0.5920179207204117
Epoch: 3| Training loss: 0.5239076409919164| Validation loss: 0.5742090121213917
Epoch: 4| Training loss: 0.500731542158432| Validation loss: 0.5599256616458367
Epoch: 5| Training loss: 0.481386614351927| Validation loss: 0.5483399203516909
Epoch: 6| Training loss: 0.46462260481387196| Validation loss: 0.536726260641603
Epoch: 7| Training loss: 0.4500982582962053| Validation loss: 0.5293935827417584
Epoch: 8| Training loss: 0.4361300598154409| Validation loss: 0.5194135185643873
Epoch: 9| Training loss: 0.42421092133995136| Validation loss: 0.5128852380111468
Epoch: 10| Training loss: 0.41424666156856305| Validation loss: 0.5081058150491683
Epoch: 11| Training loss: 0.40290405763261916| Validation loss: 0.4998331750481731
Epoch: 12| Training loss: 0.39344726405965164| Validation loss: 0.4940158979403169
Epoch: 13| Training loss: 0.38503789030766256| Validation loss: 0.48976802503845507
Epoch: 14| Training loss: 0.37654890135179875| Validation loss: 0.484037098115325
Epoch: 15| Training loss: 0.36894058100253413| Validation loss: 0.4798610532741002
Epoch: 16| Training loss: 0.3617911351634668| Validation loss: 0.47588612765070365
Epoch: 17| Training loss: 0.3551265404206662| Validation loss: 0.47234842804380855
Epoch: 18| Training loss: 0.3486529010996841| Validation loss: 0.46797755621787246
Epoch: 19| Training loss: 0.34244316989524753| Validation loss: 0.464938665019058
Epoch: 20| Training loss: 0.3366343015770723| Validation loss: 0.4616842926731529
Epoch: 21| Training loss: 0.3312023236130169| Validation loss: 0.4590262056110472
Epoch: 22| Training loss: 0.3260007588631569| Validation loss: 0.4563979800942745
Epoch: 23| Training loss: 0.32083959589960936| Validation loss: 0.45317017998003734
Epoch: 24| Training loss: 0.31705059120551277| Validation loss: 0.4527182622861437
Epoch: 25| Training loss: 0.31137245815884257| Validation loss: 0.4484012987783105
Epoch: 26| Training loss: 0.3069467760380594| Validation loss: 0.4461629481765205
Epoch: 27| Training loss: 0.30267985376747814| Validation loss: 0.4440273276849877
Epoch: 28| Training loss: 0.298560387521571| Validation loss: 0.4418603123640002
Epoch: 29| Training loss: 0.29558254986979837| Validation loss: 0.44172617207437204
Epoch: 30| Training loss: 0.29078322582916094| Validation loss: 0.4380952460056868
Epoch: 31| Training loss: 0.28718417285149966| Validation loss: 0.4362850387634425
Epoch: 32| Training loss: 0.28408560988552123| Validation loss: 0.43577239283434904
Epoch: 33| Training loss: 0.28009737012982255| Validation loss: 0.4331545678568254
Epoch: 34| Training loss: 0.2767640337246711| Validation loss: 0.431657490301419
Epoch: 35| Training loss: 0.27350820080571875| Validation loss: 0.43010420069230215
Epoch: 36| Training loss: 0.27036735251938776| Validation loss: 0.42862832552853375
Epoch: 37| Training loss: 0.2673211984298023| Validation loss: 0.42731624333970303
Epoch: 38| Training loss: 0.2643682626690211| Validation loss: 0.4260636123942821
Epoch: 39| Training loss: 0.2615143553313424| Validation loss: 0.4247602930846561
Epoch: 40| Training loss: 0.25872058410623594| Validation loss: 0.4235425412009785
Epoch: 41| Training loss: 0.2560177969586485| Validation loss: 0.42236637774851515
Epoch: 42| Training loss: 0.2533546845784477| Validation loss: 0.42128448240471866
Epoch: 43| Training loss: 0.2507883305559025| Validation loss: 0.4202410832783033
Epoch: 44| Training loss: 0.24825701835297637| Validation loss: 0.41923554264174134
Epoch: 45| Training loss: 0.2458218689350721| Validation loss: 0.41821537954238325
Epoch: 46| Training loss: 0.2435181956552816| Validation loss: 0.4175138038837945
Epoch: 47| Training loss: 0.241183986857215| Validation loss: 0.4165877792887614
```

```
Epoch: 48| Training loss: 0.2389589545183026| Validation loss: 0.41557310746919895
Epoch: 49| Training loss: 0.23680171011929335| Validation loss: 0.4147816507167802
Epoch: 50| Training loss: 0.23450601755766678| Validation loss: 0.41387587000162357
Epoch: 51| Training loss: 0.23247124983002218| Validation loss: 0.4131240893983248
Epoch: 52| Training loss: 0.23033786650938054| Validation loss: 0.41233527047381
Epoch: 53| Training loss: 0.2282567887396891| Validation loss: 0.41156614941290953
Epoch: 54| Training loss: 0.22647270304097009| Validation loss: 0.41099267683408564
Epoch: 55| Training loss: 0.2246358767100557| Validation loss: 0.41042708711540626
Epoch: 56| Training loss: 0.22246017419395409| Validation loss: 0.40954002861607436
Epoch: 57| Training loss: 0.22060668292241928| Validation loss: 0.40889556239058955
Epoch: 58| Training loss: 0.2187885160668603| Validation loss: 0.4082432051363766
Epoch: 59| Training loss: 0.2170224169340236| Validation loss: 0.4076163788242318
Epoch: 60| Training loss: 0.21530320470315473| Validation loss: 0.40713880258386625
Epoch: 61| Training loss: 0.21357932142988068| Validation loss: 0.4065210097418873
Epoch: 62| Training loss: 0.21191185411079208| Validation loss: 0.4060224089558166
Epoch: 63| Training loss: 0.21032271463596797| Validation loss: 0.4054921850332128
Epoch: 64| Training loss: 0.2087609426053154| Validation loss: 0.40501547722330933
Epoch: 65| Training loss: 0.2071890996095131| Validation loss: 0.4045265690542238
Epoch: 66| Training loss: 0.20560751820427933| Validation loss: 0.4040820565490143
Epoch: 67| Training loss: 0.20406864906806577| Validation loss: 0.40361262444820656
Epoch: 68| Training loss: 0.2025156464996579| Validation loss: 0.4030518995290221
Epoch: 69| Training loss: 0.2010378950218318| Validation loss: 0.4025949361740047
Epoch: 70| Training loss: 0.1997728849127573| Validation loss: 0.4023991394978512
Epoch: 71| Training loss: 0.19841801966004854| Validation loss: 0.40200966293582197
Epoch: 72| Training loss: 0.1968157840876145| Validation loss: 0.4013443623862235
Epoch: 73| Training loss: 0.19555410813697371| Validation loss: 0.4011169859186285
Epoch: 74| Training loss: 0.19413924871716362| Validation loss: 0.4006309698407786
Epoch: 75| Training loss: 0.19278085382790883| Validation loss: 0.40022613590881917
Epoch: 76| Training loss: 0.1914947308475946| Validation loss: 0.3998871578256288
Epoch: 77| Training loss: 0.190261922351623| Validation loss: 0.39960823107953586
Epoch: 78| Training loss: 0.18896911417129283| Validation loss: 0.39922627783002923
Epoch: 79| Training loss: 0.1877191702258717| Validation loss: 0.3988834207898679
Epoch: 80| Training loss: 0.18650134006367705| Validation loss: 0.3985725136186029
Epoch: 81| Training loss: 0.1853023306621783| Validation loss: 0.3982626757047629
Epoch: 82| Training loss: 0.18425246300322776| Validation loss: 0.3981454498168277
Epoch: 83| Training loss: 0.18298816020164818| Validation loss: 0.39772399858098695
```

Now plot the training and validation history per epoch. Does your model underfit, overfit or is it about right? Explain why.

In [27]:



Explain here...
Compute accuracy, precision, recall and F1-scores:

```
In [10]: # fill in your code...

         print('Accuracy:', accuracy_score(Y_te,preds_te_count))
         print('Precision:', precision_score(Y_te,preds_te_count))
         print('Recall:', recall_score(Y_te,preds_te_count))
         print('F1-Score:', f1_score(Y_te,preds_te_count))
```

Finally, print the top-10 words for the negative and positive class respectively.

```
In [8]: # fill in your code...

In [9]: # fill in your code...
```

If we were to apply the classifier we've learned into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain?
Provide your answer here...

3.2 Train and Evaluate Logistic Regression with TF.IDF vectors

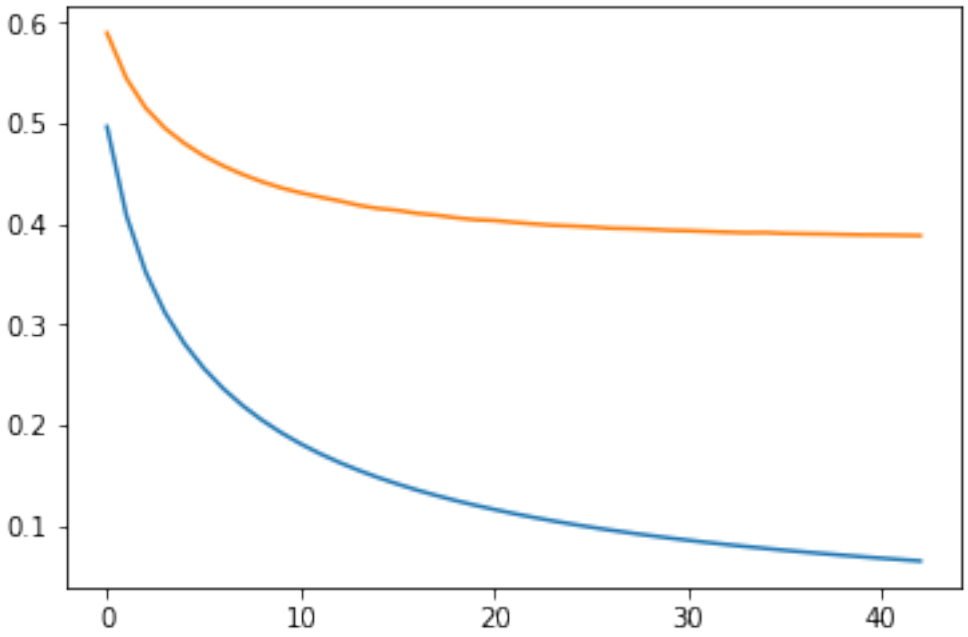
Follow the same steps as above (i.e. evaluating count n-gram representations).

```
In [31]: w_tfidf, trl, devl = SGD(X_tr_tfidf, Y_tr,
                                  X_dev=X_dev_tfidf,
                                  Y_dev=Y_dev,
                                  lr=0.0001,
                                  alpha=0.00001,
                                  epochs=50)

Epoch: 0| Training loss: 0.496538116279205| Validation loss: 0.5895899274588817
Epoch: 1| Training loss: 0.4084394710560143| Validation loss: 0.544649086837174


/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:50: DeprecationWarning: elementwis


Epoch: 2| Training loss: 0.35196590407420064| Validation loss: 0.5150876603863132
Epoch: 3| Training loss: 0.3116844242022623| Validation loss: 0.49489773731674874
Epoch: 4| Training loss: 0.2810715774838783| Validation loss: 0.47997275137676726
Epoch: 5| Training loss: 0.25668747878708276| Validation loss: 0.46763654113258013
Epoch: 6| Training loss: 0.23654276549913594| Validation loss: 0.45767733688153556
Epoch: 7| Training loss: 0.21965240746100156| Validation loss: 0.4492067736453032
Epoch: 8| Training loss: 0.20518502986248432| Validation loss: 0.44195046817180716
Epoch: 9| Training loss: 0.1926379393530923| Validation loss: 0.4358175343417367
Epoch: 10| Training loss: 0.18161013121343478| Validation loss: 0.430927744935762
Epoch: 11| Training loss: 0.17183241812407454| Validation loss: 0.4267194647775054
Epoch: 12| Training loss: 0.1631180196151434| Validation loss: 0.42287948202462305
Epoch: 13| Training loss: 0.1552643554313739| Validation loss: 0.4185174243921952
Epoch: 14| Training loss: 0.14816016424795736| Validation loss: 0.41535439115243267
Epoch: 15| Training loss: 0.14171645583977022| Validation loss: 0.41354500028973346
Epoch: 16| Training loss: 0.13573592746279245| Validation loss: 0.41050631949541183
Epoch: 17| Training loss: 0.13034674080565548| Validation loss: 0.40873127609435583
Epoch: 18| Training loss: 0.12530074212468914| Validation loss: 0.4062032520033557
Epoch: 19| Training loss: 0.12068043847628 21| Validation loss: 0.4042355558277022
Epoch: 20| Training loss: 0.11641749849268634| Validation loss: 0.40343553356873935
Epoch: 21| Training loss: 0.11238278831491308| Validation loss: 0.4016416833615827
Epoch: 22| Training loss: 0.10865156439664284| Validation loss: 0.4000595888878182
Epoch: 23| Training loss: 0.10516197433948125| Validation loss: 0.39868784044604255
Epoch: 24| Training loss: 0.10189115553121805| Validation loss: 0.3979524616390154
Epoch: 25| Training loss: 0.09882058648610967| Validation loss: 0.3970440497535089
Epoch: 26| Training loss: 0.09592138473665975| Validation loss: 0.395778881579832
Epoch: 27| Training loss: 0.09320843385055887| Validation loss: 0.3953557357339525
Epoch: 28| Training loss: 0.09063019016289853| Validation loss: 0.39470595909680056
Epoch: 29| Training loss: 0.08817629918827585| Validation loss: 0.3937076446246101
Epoch: 30| Training loss: 0.08586942642600201| Validation loss: 0.3932263994003147
Epoch: 31| Training loss: 0.08367305497260015| Validation loss: 0.3925446707702044
Epoch: 32| Training loss: 0.08158397572757985| Validation loss: 0.39185414682427805
Epoch: 33| Training loss: 0.0795957520805118| Validation loss: 0.3912667860138064
Epoch: 34| Training loss: 0.0777226931855252| Validation loss: 0.3914079793827536
Epoch: 35| Training loss: 0.07590268082740566| Validation loss: 0.39055218226682514
```

```
Epoch: 36| Training loss: 0.07417745810765715| Validation loss: 0.39023461847996976
Epoch: 37| Training loss: 0.07253087010654821| Validation loss: 0.3899764154646073
Epoch: 38| Training loss: 0.07095338295428982| Validation loss: 0.38952321238763404
Epoch: 39| Training loss: 0.06944490333053023| Validation loss: 0.3891662489075472
Epoch: 40| Training loss: 0.06799534146111857| Validation loss: 0.389052493545359
Epoch: 41| Training loss: 0.06660689466800641| Validation loss: 0.3887629118157739
Epoch: 42| Training loss: 0.06527371405647933| Validation loss: 0.38843451100305176
```

Now plot the training and validation history per epoch. Does your model underfit, overfit or is it about right? Explain why.

```
In [32]: # fill in your code...
```



Compute accuracy, precision, recall and F1-scores:

```
In [11]: # fill in your code...

         print('Accuracy:', accuracy_score(Y_te,preds_te))
         print('Precision:', precision_score(Y_te,preds_te))
         print('Recall:', recall_score(Y_te,preds_te))
         print('F1-Score:', f1_score(Y_te,preds_te))
```

Print top-10 most positive and negative words:

```
In [13]: # fill in your code...

In [14]: # fill in your code...
```

3.2.1 Discuss how did you choose model hyperparameters (e.g. learning rate and regularisation strength)? What is the relation between training epochs and learning rate? How the regularisation strength affects performance?

Enter your answer here...

3.3 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count			
BOW-tfidf			

4 Multi-class Logistic Regression

Now you need to train a Multiclass Logistic Regression (MLR) Classifier by extending the Binary model you developed above. You will use the MLR model to perform topic classification on the AG news dataset consisting of three classes:

- Class 1: World
- Class 2: Sports
- Class 3: Business

You need to follow the same process as in Task 1 for data processing and feature extraction by reusing the functions you wrote.

```
In [36]: # fill in your code...

In [37]: data_tr.head()
```

Out[37]:

	label	text
0	1	Reuters - Venezuelans turned out early\and in ...
1	1	Reuters - South Korean police used water canno...
2	1	Reuters - Thousands of Palestinian\prisoners i...
3	1	AFP - Sporadic gunfire and shelling took place...
4	1	AP - Dozens of Rwandan soldiers flew into Suda...

```
In [38]: # fill in your code...

In [39]: vocab, df, ngram_counts = get_vocab(X_tr_raw, ngram_range=(1,3), keep_topN=5000, stop_
         print(len(vocab))
         print()
         print(list(vocab)[:100])
         print()
         print(df.most_common()[:10])
```

```
5000
```

```
['questions', ('exhibition', 'game'), ('computer', 'maker'), ('chavez', 'won'), ('invoices', '
```

```
[('reuters', 631), ('said', 432), ('tuesday', 413), ('wednesday', 344), ('new', 325), ('after'
```

```
In [40]: # fill in your code...
```

Now you need to change `SGD` to support multiclass datasets. First you need to develop a `softmax` function. It takes as input:

- `z`: array of real numbers

and returns:

- `smax`: the softmax of `z`

```
In [42]: def softmax(z):

             # fill in your code...

             return smax
```

Then modify `predict_proba` and `predict_class` functions for the multiclass case:

```
In [43]: def predict_proba(X, weights):

             # fill in your code...

             return preds_proba
```

```
In [44]: def predict_class(X, weights):

             # fill in your code...

             return preds_class
```

Toy example and expected functionality of the functions above:

```
In [45]: X = np.array([[0.1,0.2],[0.2,0.1],[0.1,-0.2]])
         w = np.array([[2,-5],[-5,2]])
```

```
In [46]: predict_proba(X, w)
```

```
Out[46]: array([[0.33181223, 0.66818777],
                [0.66818777, 0.33181223],
                [0.89090318, 0.10909682]])
```

```
In [47]: predict_class(X, w)
```

```
Out[47]: array([2, 1, 1])
```

Now you need to compute the categorical cross entropy loss (extending the binary loss to support multiple classes).

```
In [1]: def categorical_loss(X, Y, weights, num_classes=5, alpha=0.00001):

            # fill in your code...

            return l
```

Finally you need to modify SGD to support the categorical cross entropy loss:

```
In [49]: def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], num_classes=5, lr=0.01, alpha=0.00001, epochs=

             # fill in your code...

             return weights, training_loss_history, validation_loss_history
```

Now you are ready to train and evaluate you MLR following the same steps as in Task 1 for both Count and tfidf features:

```
In [50]: w_count, loss_tr_count, dev_loss_count = SGD(X_tr_count, Y_tr,
                                                      X_dev=X_dev_count,
                                                      Y_dev=Y_dev,
                                                      num_classes=3,
                                                      lr=0.0001,
                                                      alpha=0.001,
                                                      epochs=200)

/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:55: DeprecationWarning: elementwis


Epoch: 0| Training loss: 1.0797445295473487| Validation loss: 1.081372090349833
Epoch: 1| Training loss: 1.079316160352198| Validation loss: 1.0640685482079149
Epoch: 2| Training loss: 1.0560473439817515| Validation loss: 1.0466716664674833
Epoch: 3| Training loss: 1.0823272116547453| Validation loss: 1.0293144599016153
Epoch: 4| Training loss: 1.0541411400498168| Validation loss: 1.0120430537618956
Epoch: 5| Training loss: 0.6018201485747277| Validation loss: 0.9949501699020834
Epoch: 6| Training loss: 0.9408020769124341| Validation loss: 0.9780518842493735
Epoch: 7| Training loss: 0.9785218029526777| Validation loss: 0.961399507260303
Epoch: 8| Training loss: 1.105504123262664| Validation loss: 0.945017429821232
Epoch: 9| Training loss: 0.9330426453622344| Validation loss: 0.9289397744769093
Epoch: 10| Training loss: 0.7202271854150083| Validation loss: 0.9131712383598369
Epoch: 11| Training loss: 0.980165568395653| Validation loss: 0.8977588631301825
Epoch: 12| Training loss: 0.9350596877302814| Validation loss: 0.8826257874396408
Epoch: 13| Training loss: 1.015964417226491| Validation loss: 0.8678640000045689
Epoch: 14| Training loss: 0.7853481727623568| Validation loss: 0.8534092079475973
```

```
Epoch: 15| Training loss: 1.1280839072469695| Validation loss: 0.8392859262760239
Epoch: 16| Training loss: 0.45596125168525925| Validation loss: 0.8254557688887008
Epoch: 17| Training loss: 1.0070191455181927| Validation loss: 0.8120053087983156
Epoch: 18| Training loss: 0.9938827745552048| Validation loss: 0.7989146470042358
Epoch: 19| Training loss: 0.5941619165230784| Validation loss: 0.7861148779646865
Epoch: 20| Training loss: 0.347645483062249| Validation loss: 0.773639617757056
Epoch: 21| Training loss: 0.8615779910495689| Validation loss: 0.7614413103814573
Epoch: 22| Training loss: 0.8559620095999312| Validation loss: 0.7495740843572936
Epoch: 23| Training loss: 0.7123026165421295| Validation loss: 0.7380118770283757
Epoch: 24| Training loss: 0.628420348319027| Validation loss: 0.7267497405327836
Epoch: 25| Training loss: 1.1143453645268255| Validation loss: 0.7157650456550849
Epoch: 26| Training loss: 0.5053554837538091| Validation loss: 0.7050752398160355
Epoch: 27| Training loss: 0.8580561710864723| Validation loss: 0.6946265481406195
Epoch: 28| Training loss: 1.0145821027394442| Validation loss: 0.6844644903515154
Epoch: 29| Training loss: 0.889803100593025| Validation loss: 0.6745416521972386
Epoch: 30| Training loss: 0.3512373705994844| Validation loss: 0.6649076709998702
Epoch: 31| Training loss: 0.6946077278013651| Validation loss: 0.6554896489368593
Epoch: 32| Training loss: 0.5721038746595615| Validation loss: 0.6463159196618706
Epoch: 33| Training loss: 0.9286421195752351| Validation loss: 0.6374037446331372
Epoch: 34| Training loss: 0.3043301520488227| Validation loss: 0.6286933702647342
Epoch: 35| Training loss: 0.8412715058863925| Validation loss: 0.6201950715638889
Epoch: 36| Training loss: 1.0162421996444875| Validation loss: 0.6119312128918258
Epoch: 37| Training loss: 0.9800875509382644| Validation loss: 0.6038364508265476
Epoch: 38| Training loss: 0.2529787526410042| Validation loss: 0.5959680427407086
Epoch: 39| Training loss: 0.5204508487025243| Validation loss: 0.5882845644274958
Epoch: 40| Training loss: 0.663760559879069| Validation loss: 0.5807678321194896
Epoch: 41| Training loss: 0.15865889006553835| Validation loss: 0.5734687699728561
Epoch: 42| Training loss: 0.6408607725036662| Validation loss: 0.5663184689700342
Epoch: 43| Training loss: 0.2797611441024611| Validation loss: 0.5593606258380786
Epoch: 44| Training loss: 0.7852857528209075| Validation loss: 0.552554302011612
Epoch: 45| Training loss: 1.006186400841472| Validation loss: 0.5459031927551995
Epoch: 46| Training loss: 1.121242812418955| Validation loss: 0.5394152242463135
Epoch: 47| Training loss: 0.45275382650303764| Validation loss: 0.5330716806209664
Epoch: 48| Training loss: 0.5002302346473977| Validation loss: 0.5268516486205744
Epoch: 49| Training loss: 0.2927756479030764| Validation loss: 0.520796556633722
Epoch: 50| Training loss: 0.5815213659515958| Validation loss: 0.514875429745833
Epoch: 51| Training loss: 0.43511107264972204| Validation loss: 0.509093372083549
Epoch: 52| Training loss: 0.2004456471210612| Validation loss: 0.5034369186797815
Epoch: 53| Training loss: 0.6927294724851424| Validation loss: 0.4978805528962536
Epoch: 54| Training loss: 0.22762255875684806| Validation loss: 0.49248110263913736
Epoch: 55| Training loss: 0.41286842099956444| Validation loss: 0.4871768560097116
Epoch: 56| Training loss: 0.2858867912825694| Validation loss: 0.4820023098168036
Epoch: 57| Training loss: 1.07945594292479| Validation loss: 0.4769355688611146
Epoch: 58| Training loss: 0.08632099484327364| Validation loss: 0.4719707627364577
Epoch: 59| Training loss: 0.4380467526003749| Validation loss: 0.4671259629523961
Epoch: 60| Training loss: 0.4504033371479604| Validation loss: 0.4623682646862233
Epoch: 61| Training loss: 0.8991535203093632| Validation loss: 0.45772777193305997
Epoch: 62| Training loss: 0.37976053429018847| Validation loss: 0.453170172687834
```

```
Epoch: 63| Training loss: 0.6690249969029779| Validation loss: 0.4487003465696454
Epoch: 64| Training loss: 0.12214122740982129| Validation loss: 0.4443374960034709
Epoch: 65| Training loss: 0.6816289535486254| Validation loss: 0.4400584436398409
Epoch: 66| Training loss: 0.18194708158942638| Validation loss: 0.4358569111132549
Epoch: 67| Training loss: 0.5175968359577696| Validation loss: 0.4317374156856518
Epoch: 68| Training loss: 0.2748850713851889| Validation loss: 0.42768780305615756
Epoch: 69| Training loss: 0.47027298142101503| Validation loss: 0.42374507168474584
Epoch: 70| Training loss: 0.19929430369829634| Validation loss: 0.4198554513354222
Epoch: 71| Training loss: 0.3119339792974164| Validation loss: 0.4160593000805275
Epoch: 72| Training loss: 0.2802330752147449| Validation loss: 0.4123300080340496
Epoch: 73| Training loss: 1.4068722074169777| Validation loss: 0.4086685670109219
Epoch: 74| Training loss: 1.2289700039889522| Validation loss: 0.4050692506591351
Epoch: 75| Training loss: 0.5879134607806588| Validation loss: 0.4015404749982959
Epoch: 76| Training loss: 0.5598034047715446| Validation loss: 0.3980768791999669
Epoch: 77| Training loss: 0.4072390166320532| Validation loss: 0.3946781657347487
Epoch: 78| Training loss: 0.4720387020720993| Validation loss: 0.39135074144554305
Epoch: 79| Training loss: 0.8652127247147093| Validation loss: 0.3880767411953969
Epoch: 80| Training loss: 0.49502493338551296| Validation loss: 0.38485336346274324
Epoch: 81| Training loss: 0.8478719397159928| Validation loss: 0.3816873793123302
Epoch: 82| Training loss: 0.5769432672302212| Validation loss: 0.3785768379529342
Epoch: 83| Training loss: 0.25865075231928386| Validation loss: 0.37552080913970615
Epoch: 84| Training loss: 0.6780205857236974| Validation loss: 0.37251982296888037
Epoch: 85| Training loss: 0.5717729327147981| Validation loss: 0.36958152892125545
Epoch: 86| Training loss: 0.7185798657407652| Validation loss: 0.36668874595841955
Epoch: 87| Training loss: 0.32730720069458585| Validation loss: 0.3638467143325988
Epoch: 88| Training loss: 0.9394375601966297| Validation loss: 0.3610402229621626
Epoch: 89| Training loss: 0.5314941113071576| Validation loss: 0.3582963265282113
Epoch: 90| Training loss: 0.9159430629959491| Validation loss: 0.3556047452606419
Epoch: 91| Training loss: 0.27830594341362347| Validation loss: 0.3529452396878083
Epoch: 92| Training loss: 0.08922609875671272| Validation loss: 0.3503285390711649
Epoch: 93| Training loss: 0.5109477286610166| Validation loss: 0.34776466510396653
Epoch: 94| Training loss: 0.8247015373951916| Validation loss: 0.34523113629834656
Epoch: 95| Training loss: 1.4111970700349956| Validation loss: 0.3427459408526143
Epoch: 96| Training loss: 0.9973410892711908| Validation loss: 0.3403031775623844
Epoch: 97| Training loss: 0.9298826919970149| Validation loss: 0.3378961810444463
Epoch: 98| Training loss: 0.14036615190332108| Validation loss: 0.33552333229158426
Epoch: 99| Training loss: 0.42857059017867416| Validation loss: 0.3331908153000918
Epoch: 100| Training loss: 0.5909452468922335| Validation loss: 0.3308943792523601
Epoch: 101| Training loss: 0.0582537404726638| Validation loss: 0.3286410364885621
Epoch: 102| Training loss: 0.8460500582683682| Validation loss: 0.32641969802577286
Epoch: 103| Training loss: 0.5298538159246153| Validation loss: 0.324225708977352
Epoch: 104| Training loss: 0.3002611196487631| Validation loss: 0.3220709147432522
Epoch: 105| Training loss: 0.6857676458335961| Validation loss: 0.31995106501678755
Epoch: 106| Training loss: 0.5195060392225062| Validation loss: 0.31785403975508386
Epoch: 107| Training loss: 0.6192846034415994| Validation loss: 0.3157920207863501
Epoch: 108| Training loss: 0.09917322459957126| Validation loss: 0.31376264796419107
Epoch: 109| Training loss: 0.3136491065708976| Validation loss: 0.3117644717911298
Epoch: 110| Training loss: 0.2961228303588815| Validation loss: 0.30979000698586306
```

```
Epoch: 111| Training loss: 0.028415357821764815| Validation loss: 0.30785497759094477
Epoch: 112| Training loss: 0.3321063258469021| Validation loss: 0.3059320900816925
Epoch: 113| Training loss: 0.2679899421174463| Validation loss: 0.30404511603354395
Epoch: 114| Training loss: 0.8990533088234927| Validation loss: 0.30218664002975204
Epoch: 115| Training loss: 0.12905768556887146| Validation loss: 0.3003505904610916
Epoch: 116| Training loss: 0.4506684356556081| Validation loss: 0.2985291037359777
Epoch: 117| Training loss: 0.844468526893116| Validation loss: 0.29674277317528236
Epoch: 118| Training loss: 0.6767336880461876| Validation loss: 0.2949822521732923
Epoch: 119| Training loss: 0.01799443908408846| Validation loss: 0.29324461004936575
Epoch: 120| Training loss: 0.3418078466994673| Validation loss: 0.2915307136152019
Epoch: 121| Training loss: 0.6270705561081482| Validation loss: 0.28984127182168334
Epoch: 122| Training loss: 1.1478485044448523| Validation loss: 0.2881789628649678
Epoch: 123| Training loss: 0.2949766090593473| Validation loss: 0.2865338197457586
Epoch: 124| Training loss: 1.4238940813590628| Validation loss: 0.2849148292705895
Epoch: 125| Training loss: 0.5994789520817773| Validation loss: 0.28331862128435503
Epoch: 126| Training loss: 0.24891273419811613| Validation loss: 0.2817370423727986
Epoch: 127| Training loss: 0.39238436079518574| Validation loss: 0.2801812115027427
Epoch: 128| Training loss: 1.5600414187349476| Validation loss: 0.27864540709644464
Epoch: 129| Training loss: 0.9472198772310756| Validation loss: 0.27712518454869994
Epoch: 130| Training loss: 0.1584699773317698| Validation loss: 0.2756290475247621
Epoch: 131| Training loss: 0.20644196621222938| Validation loss: 0.2741530162974089
Epoch: 132| Training loss: 0.18215283594621412| Validation loss: 0.2726944798834253
Epoch: 133| Training loss: 0.8414874757451525| Validation loss: 0.2712551700923043
Epoch: 134| Training loss: 0.37457828472756166| Validation loss: 0.269827937621642
Epoch: 135| Training loss: 0.00774395687073208| Validation loss: 0.26842030110408316
Epoch: 136| Training loss: 0.75978129405607| Validation loss: 0.2670359677369942
Epoch: 137| Training loss: 0.04094798273170632| Validation loss: 0.2656598668535911
Epoch: 138| Training loss: 1.1939040225650244| Validation loss: 0.2642977146554992
Epoch: 139| Training loss: 0.2706364434128102| Validation loss: 0.26296275384475015
Epoch: 140| Training loss: 0.4041706720632301| Validation loss: 0.2616457843006157
Epoch: 141| Training loss: 0.3930983060172604| Validation loss: 0.2603420380125703
Epoch: 142| Training loss: 0.08588635860652719| Validation loss: 0.2590556686494602
Epoch: 143| Training loss: 0.24441280871964122| Validation loss: 0.25778064807349627
Epoch: 144| Training loss: 0.8777114045822041| Validation loss: 0.25652165984926234
Epoch: 145| Training loss: 0.8194686345456902| Validation loss: 0.25528228171012946
Epoch: 146| Training loss: 0.5319086987289504| Validation loss: 0.2540533247787649
Epoch: 147| Training loss: 0.09005366358246589| Validation loss: 0.2528353480461422
Epoch: 148| Training loss: 0.21640847881454545| Validation loss: 0.251638801334596
Epoch: 149| Training loss: 0.14749728182299668| Validation loss: 0.2504513185602062
Epoch: 150| Training loss: 0.9646686003838391| Validation loss: 0.249274942485048
Epoch: 151| Training loss: 0.8459778376210175| Validation loss: 0.24811939173669992
Epoch: 152| Training loss: 0.3289512046629609| Validation loss: 0.24697182796581005
Epoch: 153| Training loss: 0.010146911359652084| Validation loss: 0.2458389491289637
Epoch: 154| Training loss: 0.08495971132730216| Validation loss: 0.244718975192335
Epoch: 155| Training loss: 0.5977737889322374| Validation loss: 0.24360572221188478
Epoch: 156| Training loss: 0.0851881124765378| Validation loss: 0.24251352180512406
Epoch: 157| Training loss: 0.8071434173631403| Validation loss: 0.2414341422328391
Epoch: 158| Training loss: 0.16374943118546553| Validation loss: 0.24036653931680665
```

```
Epoch: 159| Training loss: 0.19922472773981986| Validation loss: 0.23930460979725618
Epoch: 160| Training loss: 0.2204896358048265| Validation loss: 0.23826151753367814
Epoch: 161| Training loss: 0.4573968802025281| Validation loss: 0.23722811498264912
Epoch: 162| Training loss: 0.1810635314345012| Validation loss: 0.23620717993755244
Epoch: 163| Training loss: 0.5517398742853074| Validation loss: 0.2351936795375895
Epoch: 164| Training loss: 0.061671999023491436| Validation loss: 0.234190439908011
```

Plot training and validation process and explain if your model overfit, underfit or is about right:

```
In [2]: # fill in your code...
```

Compute accuracy, precision, recall and F1-scores:

```
In [6]: # fill in your code...

        print('Accuracy:', accuracy_score(Y_te,preds_te))
        print('Precision:', precision_score(Y_te,preds_te,average='macro'))
        print('Recall:', recall_score(Y_te,preds_te,average='macro'))
        print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

Print the top-10 words for each class respectively.

```
In [7]: # fill in your code...
```

4.0.1 Discuss how did you choose model hyperparameters (e.g. learning rate and regularisation strength)? What is the relation between training epochs and learning rate? How the regularisation strength affects performance?

Explain here...

4.0.2 Now evaluate BOW-tfidf...

4.1 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count			
BOW-tfidf			

```
In [ ]:
```